N73-19217

AUTODOCUMENTATION

Jay Arnold
Computing & Software, Inc.

Currently available automated documentation systems, like the ones described at this symposium, perform the functions for which they were intended quite admirably for the most part. They have proven to be useful tools in the area of retrospective archival documentation and as aids in finding logic problems. However, several documentation needs remain that current systems do not fulfill.

As yet, systems that can recognize a program by type and categorize the process or that can describe the application for which a program was intended have not yet been developed. But both of these functions are necessary if automated documentation systems are to be used to solve some of the serious problems facing this industry.

It is fairly obvious that the industry is plagued with "specialization" and "originality" syndromes, that routine programs are written and rewritten for each new application. Lack of adequate documentation for the vast store of existing programs only serves to further aggravate these problems.

To decide whether an existing program can be used for a new application, it is helpful to know both how the program was originally used and what functional processes are contained in the program. The latter is particularly important in interdisciplinary transfers. But few programs have adequate documentation of this type. Most programmers preparing individual documentation are unable to see how their program or segments of it may be used in other areas. Therefore, the cost of determining the capabilities of programs or program segments usually precludes their use and forces the development of additional programs.

Automated documentation systems, with expanded capabilities, would provide a means of reusing existing programs by allowing a relatively inexpensive determination of program capabilities. This paper will present some comments on an approach to such a system. These comments do not describe any existing system and are presented solely with the intent of stimulating thought in the area.

The approach stems from observing human analysis of programs. Of course, developing a machine system on the basis of a human approach is not always the best or an efficient technique. However, it appears to be one approach to the problem.

Most programmers conduct an analysis using a source listing of the program, a description of the program input, and a description of the program output or, when available, a sample of the output itself. They tend to begin their analysis by studying the sample output to determine what they can of the original intent of the program and to look at the output data elements to provide them a "link" into the program.

Next, they take up the source listing to find an output statement that corresponds to the data element of interest from the output. From this point, their analysis is aimed at determining the content of the program rather than its original intent to determine whether the methods employed in the program are of use to them.

The human approach, then, usually starts with the program output, does an analysis of original intent, uses the output as the entrance to the program, and then proceeds to a content analysis. It should be possible to develop an automated system based on this approach. Basically, such a system would be an output-to-input analysis as opposed to the more common input-to-output flow analyses.

There are two general types of program output with which an automated system would have to contend, print and nonprint. Since the former presents less problems, it shall be considered first.

One goal in designing any automated system should be the minimization of requirements with which the programmer has to conform. The use of special control cards solely for the documentation system is an undesirable constraint. Even ordinary comment cards, while highly desirable in any documentation package, should not be a requirement for an automated system. One guideline, however, could be employed with minimal limitations on the programmer. That is the use of self-descriptive labels on all printed output as well as on nonprinted output where feasible. Since labeling is a relatively common practice, it should not prove to be a severe constraint to a programmer.

Most printed output contains two broad categories of information: report description (or header information) and data description (or lable information). In a majority of cases, some form of these two information types are present on printed output.

Header information usually includes project names, data-set descriptions, experiment types, calculation methods, names, places, dates, times, and other information that describes the purpose of that particular program output. It is this part of printed output that is of the greatest use in determining the original application.

The label information, on the other hand, pertains more specifically to the data that the program generates. Row and column labels indicate information about each sequence of calculations within the program. From these data, parameters that are being calculated and the elements to which they correspond can be deduced. In many cases, much application information is available on the printed output.

Generally, the analyzer processes this information about the program by some type of semantic and syntactic analysis. In the case where only limited information is present in a nonsentence structure, it is probable that semantic analysis would be predominant. Most programmers could probably deduce quite a bit about a program in a familiar application area by noting only a few keywords on a printout because the scope of the application area also limits the meaning and context of the terms which we see. At GSFC, the acronym OGO would immediately suggest a satellite rather than a Government organization. Given enough of these terms on a printout, within a limited context, the program application should be fairly accurately described.

The programs that do not produce printed output, such as sorting routines, utilities, and math function subroutines, will now be considered. For some types of nonprinted

output, the original application cannot be easily determined, but these are exactly the types of programs whose original intention is irrelevant. Since most of these cases are the general-purpose routines that can be used for almost any application, the decision as to whether they can be used in a new application does not depend on the original intent. However, some types of more specialized programs do not produce printed output where information pertaining to the original application would be beneficial. Even in the case of nonprinted output, three sources of information might be available: descriptions of output data sets, labels on the data sets, and cross-reference information available from printed data sets.

Probably the best place to look for the original application is at the descriptions of the output data sets. These descriptions, such as those contained in IBM 360 job control language, provide information about the size, type, and organization of the intended output. In many cases this provides clues to the application that could not be obtained from the program itself. A second source of information might be the labels on the data sets themselves. However, this information is not always available to the analyzer. When available, it can provide additional descriptive information about the original output intent. One last method of obtaining information about nonprinted outputs is to cross-reference it to information on a printed data set. In some cases, the application of the nonprinted output data set can be deduced from information on the printed output.

These are some of the means the analyzer has to deduce the original intent of the program. His approach to understanding the content or functioning of the program should be the next topic. Most computer programs, especially scientific ones, are, for the most part, a heterogeneous collection of calculations or data-manipulating processes applied to a problem area. Unfortunately, a program is usually considered as a single entity, rather than as the sum of its parts, which tends to distort the programmer's view of the inherent capability and usefulness of the parts of the program.

A brief look at any program will show that each output data element is produced by a unique sequence or "pattern" of calculations or processes. While it is true that many of these processes may overlap in multioutput programs and that some may be prerequisite to others, each output element can be traced back through the program to yield a unique pattern. Each of these processing sequences could, in most cases, be separated from the program and become an independent module with an identity and function of its own. Thus, the analyzer's task of determining program content is reduced to several subtasks of determining each of the patterns that yields an output. He must enter the program at each output data item to ascertain the pattern of processes that led to it.

As the analyzer traces the sequence of calculations and the pattern begins to clarify, he attempts to compare the developing pattern with those with which he is familiar. Unfortunately, the wide variation in programming techniques that can be employed to implement a particular well-defined process precludes the possibility of a simple comparative process. Each programmer may have a unique way of translating an established technique into coding. But while the latitude is wide, it is still finite. He is limited by the language syntax as to how he may code this process.

The analyzer must therefore be equipped to recognize the pattern from some finite range of pattern variations. To accomplish this, he must either be familiar with the entire

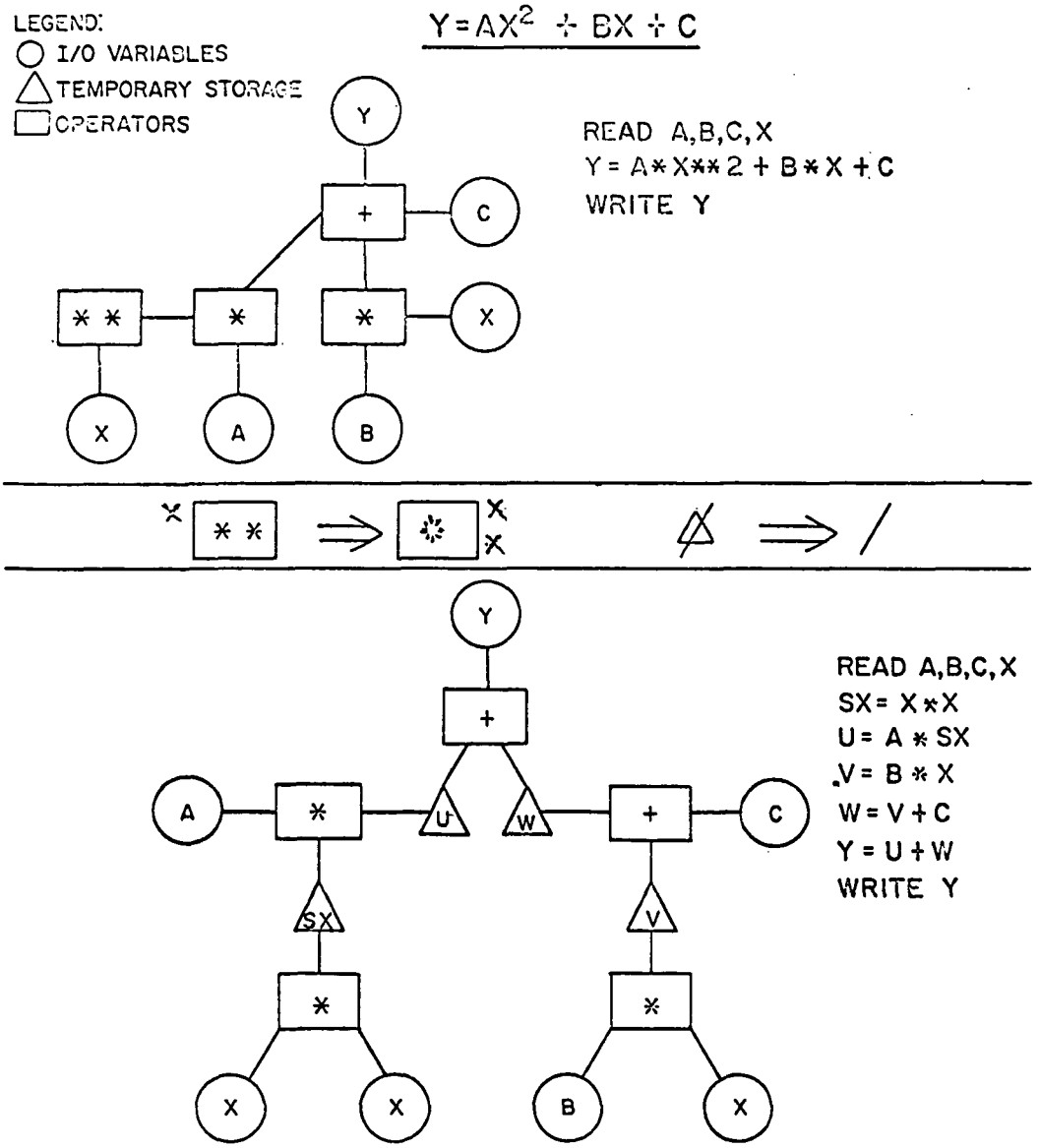

Figure 1.–Tree-structure variations.

range, which is certainly possible in many cases, or he must be capable of reducing the pattern before him to a familiar one. Whereas the latter tends to be a more difficult task, it has the advantage of requiring a familiarity with only one variation for each pattern, an advantage that might prove significant for an automated system.

While it is difficult to say exactly how a person organizes a pattern in his mind for recognition, a fair analogy might be the tree-structure representation, as indicated in figure 1. This technique has been chosen to represent patterns for two reasons: The structure can be built one level at a time, much like a human analyzer, and it is readily amenable to automated

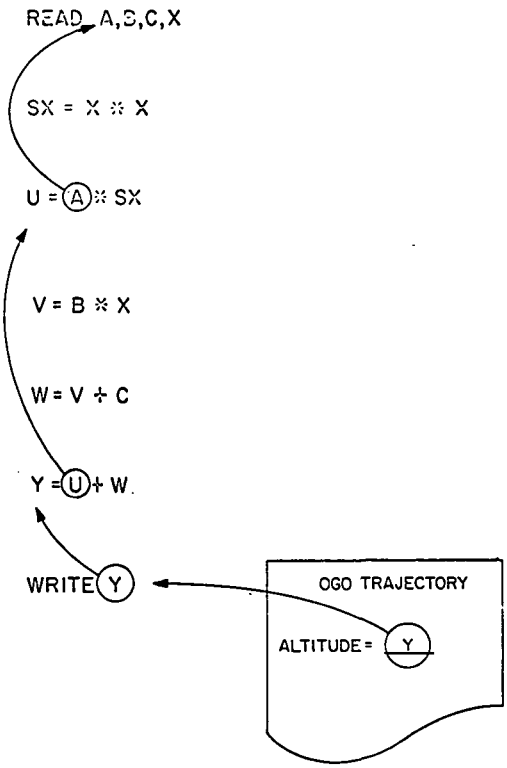

Figure 2.—Assignment statements.

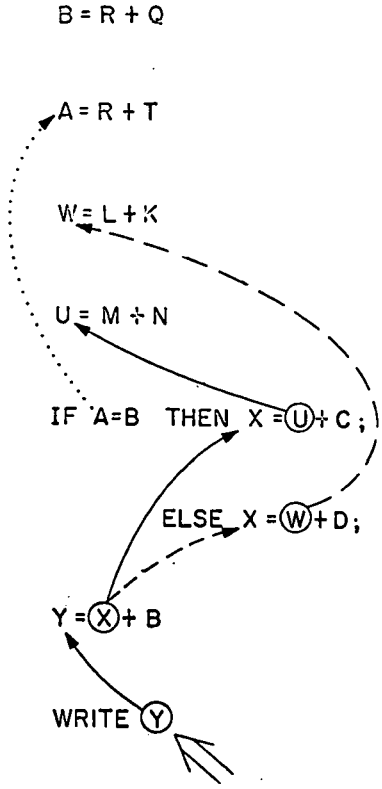

Figure 3.—Conditional assignment.

processing. As the example illustrates, the same mathematical function coded in different ways initially produces different structures. However, by applying simple techniques, the patterns can be shown to be identical.

The components of these patterns will now be considered. Although the nature of program analysis makes it, in some respects, language dependent, some of the more common general aspects can be discussed.

Once again, analysis begins with program output. The analyzer has found a data item in the output and has sought out the corresponding output statement in the program. He has identified the variable that corresponds to the data item and is about to trace the pattern of calculations.

Figure 2 shows a program consisting solely of assignment statements. The pattern is traced from the output statement to the left side of an assignment. From there, each of the preceding variables is traced to its origin, either an input or generation point. Each of the variables and operators encountered can be stored in a tree format such as the one in figure 1.

Assignment statements, though plentiful in programs, would probably be the simplest to analyze in an automated system. Although no analysis details have been worked out for the more complex processes, brief comment on some of the most common is possible.

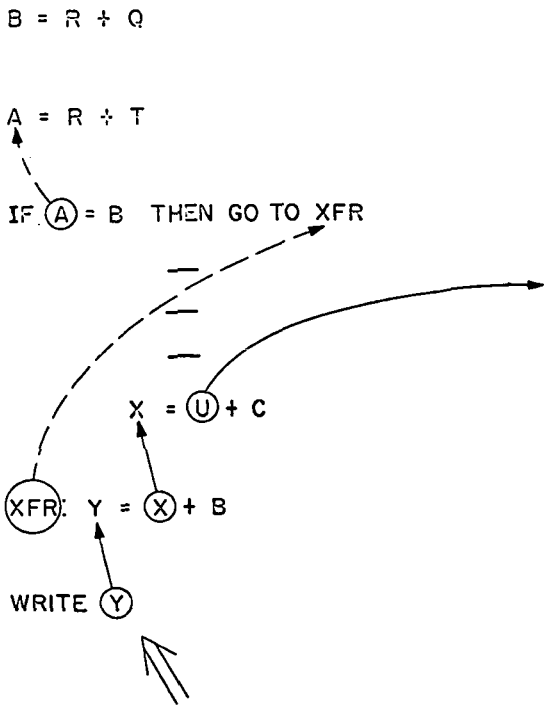

Figure 4.—Conditional branch.

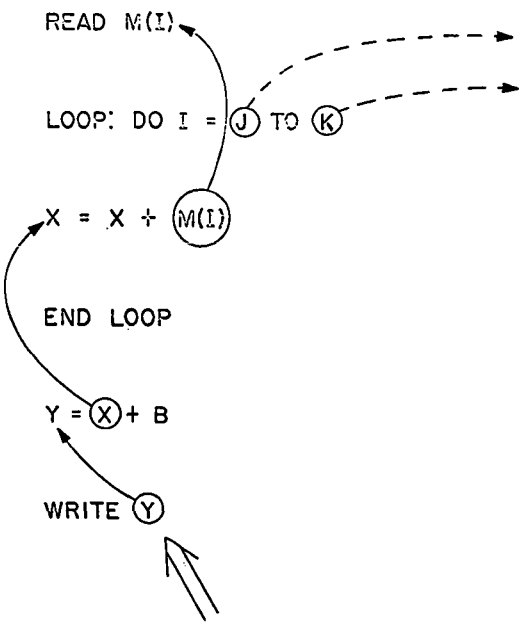

Figure 5.—Loop.

Conditional processes are probably the most common of those that might be difficult to analyze. This class of processes includes such types as conditional assignments, conditional branches, and loops.

First, the analysis of conditional assignments should be considered (fig. 3). Here one of two assignment paths can be followed depending upon the validity of the condition. The proper analysis of this statement would require tracing the pattern for both alternatives and also the pattern leading to the condition. Analysis of these three patterns would yield a complete picture of the structure of this segment. One thing this analysis might have indicated was that these were two discrete patterns, one or the other of which was selected on the basis of input data.

Next, the conditional branch shall be considered (fig. 4). In this case, the execution of an entire sequence is dependent upon the condition. If there were a direct assignment path from the output variable to the labeled statement, the presence or absence of that output would be determined by the condition. Here, analysis would require a trace of both the variable and the condition structure.

Finally, loops should be considered. In figure 5, a straight trace path is interrupted by one. To analyze this pattern, the number of iterations and nature of any discontinuities must be known, especially those near either end of the iterations. If a manual analysis were being performed, the flow of the loop at its first iteration, its second, the next to the last, and the last would be determined. Of course, if there were a conditional branch out of the loop, rather than a fixed number of iterations, that would have to be taken into consideration.

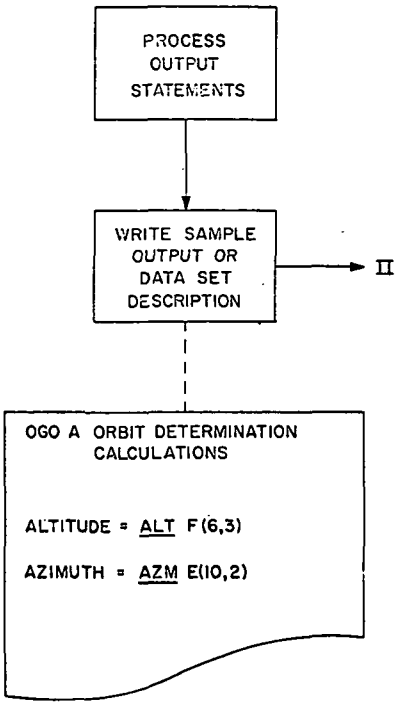

Figure 6.—Step 1 of an automated analyzer: Sample output.

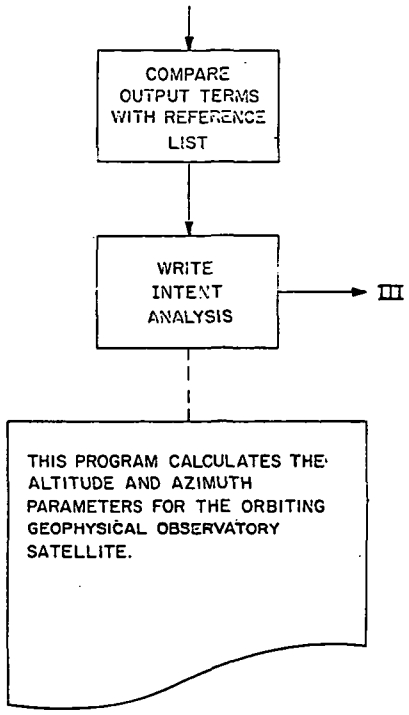

Figure 7.—Step 2 of an automated analyzer: Semantic intent analysis.

One other interesting complexity exists in this example, the presence of the same variable on both sides of an assignment. This should pose no problem in a straight assignment analysis; however, in a loop, if there were no prior reference to the variable outside, it would have to be treated as a generating point.

An automated system similiar to the one described in this paper would combine the following four previously defined functions: (1) a description of each output data set, (2) a description of the original intent of the program by analysis of the output terminology, (3) an analysis of each output-producing module, and (4) a description of all input data.

The system, after scanning the source deck of a program (fig. 6), would first process all output-related statements. From these it would produce a sample of each printed output in the program, replacing the name of the variable and its format for each output data item. For nonprinted output, the system would produce a description of the data set.

From the output statements, the system would also retrieve all significant label terms (fig. 7). It would compare these against a dictionary of terms tailored to a specific area and produce a complete description of the application of each program or segment. This part of the system could readily be merged with existing information retrieval systems, such as NASA's RECON system, which would serve as the limited-context dictionary necessary to analyze the output terminology.

This automated system would then begin an analysis of each segment of the program (fig. 8). Starting at each output data element, it would trace the pattern of calculations

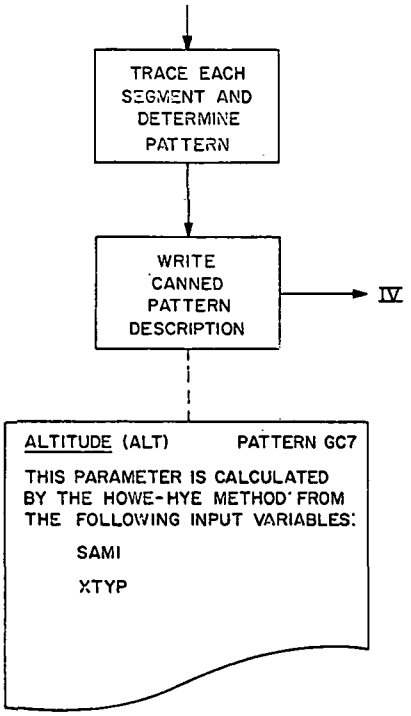

Figure 8.—Step 3 of an automated analyzer: Content analysis.

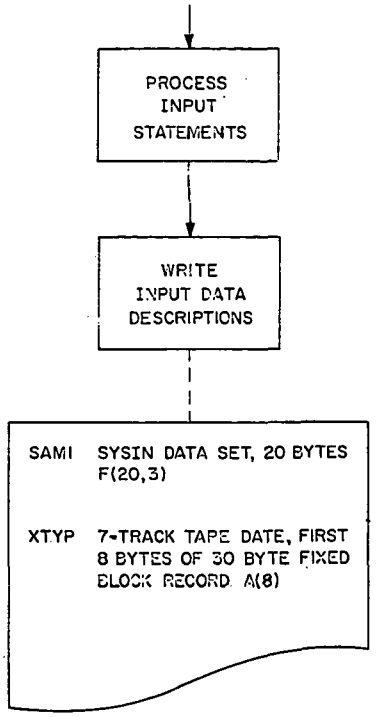

Figure 9.—Step 4 of an automated analyzer: Input data description.

using list processing techniques and reduce each to its simplest form. It would then compare these to its memory of patterns and, upon finding a match, would print prewritten descriptions of each pattern. It would also indicate what variables were input or generating points for each pattern.

Finally, it would process the input statements (fig. 9), extracting such information as it could from them in a manner similar to the processing of nonprinted output data, and would print out descriptions of each of these input items.

A system such as the one described appears to be technologically feasible now, but, to the best of my knowledge, does not exist commercially at this time. Its development, if not already under way, should soon be undertaken.

DISCUSSION

MEMBER OF THE AUDIENCE: This type of approach will only work on certain types of programs, and I was wondering if you could characterize these a little better. For example, in a simulation program, it is going to be practically impossible to trace the origin of the statistical result back through the previous simulation process. A lot of programs come in several steps in which the output is really dependent on many previous calculations that do not show up in the output. Can you characterize the type of programs that you expect us to work on?

ARNOLD: I have to admit that our original thoughts were based on scientific and business programs and not simulation programs. However, I think that it is certainly feasible to apply this to almost any class with enough effort. Although it might be difficult, it is probably within the realm of possibility to apply it even to your case.

MEMBER OF THE AUDIENCE: If you can do this, then you should also perhaps set your sights a little higher. When you accomplish this, you will also be able to verify the correctness of the programs. If you can really trace the output back through the input, you should be able to verify while you are at it, too, I should think.

ARNOLD: That would be a very excellent adjunct to such a system.

MEMBER OF THE AUDIENCE: That is a very difficult process.

ARNOLD: Yes.